Práctica de Evaluación

Máster Universitario en Ciencia de Datos e Ingeniería de Computadores

Autor
Álvaro Santana Sánchez

Extracción de características en imágenes

Extracción de Rasgos

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Dataset

Camino escogido: 2,3,4,6,7,9.

1.1. Dataset

Este ejercicio de evaluación se utiliza el dataset KMNIST [1]. Kuzushiji-MNIST es un reemplazo de MNIST con imágenes 28x28 en escala de grises. Al igual que MNIST, incluye 10 clases, provenientes de escoger un carácter por cada fila del Hiragana. Este junto al katana constituyen dos de los silabarios japoneses, los cuales son conjuntos de carácteres que simulan o aproximan sílabas [2].

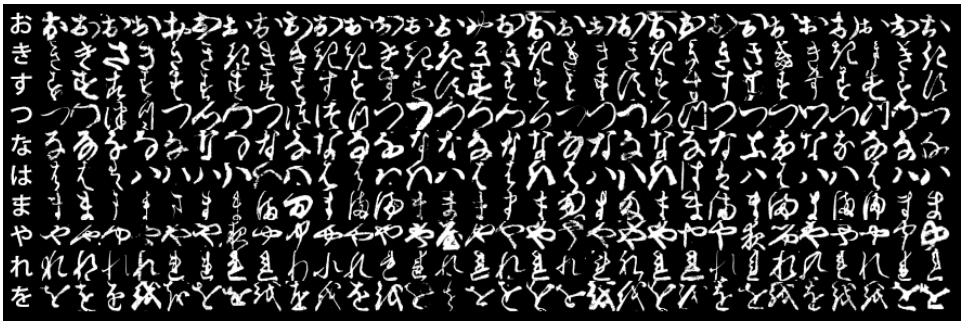

Figura 1.1: Ejemplos de carácteres Kmnist

Los carácteres escogidos para realizar el dataset pueden verse reflejados en la figura 1.1. Cada una de las filas representa un carácter diferente. Como se puede observar la homogeneidad de las clases varía, pues algunas pueden tener formas diferentes de representación. Esto puede alterar el ejercicio de aprendizaje de los descriptores, ya que una clase podrá estar formada por

diferentes tipos de representación y por ende una variedad de descriptores. Bajo estas premisas se cogen las clases 3 (positiva) y 7 (negativa), las cuáles presentan formas visualmente diferentes entre sí y sus representaciones dentro de la clase parecen ser similares (Ver figuras 1.2 y 1.3).

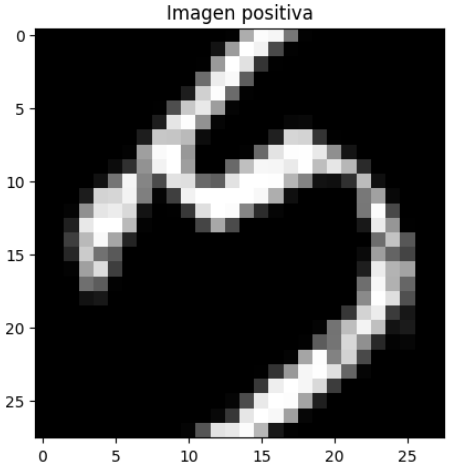

Figura 1.2: Clase 3

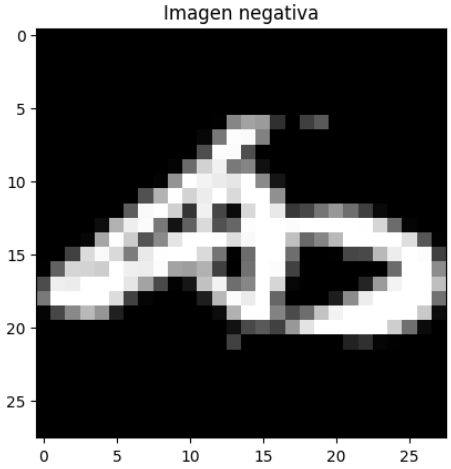

Figura 1.3: Clase 7

Capítulo 2

Histogram of Oriented Gradients.

2.0.1. Introducción

Los valores del gradiente representan el ratio de cambio de intensidad en los píxeles de una imágen, comparando cada píxel con su vecindario. Los gradientes caracterizan zonas con bordes y puntos interesantes donde la transición entre niveles de gris es significativa. Los descriptores HOG utilizan información del gradiente de cada píxel y calculan histogramas de su dirección usando bloques. Para ello, primero es calculado el gradiente en cada una de las direcciones del mismo. A partir de los gradientes en Gx y Gy se puede calcular la magnitud y dirección del gradiente a través de una transformación de coordenadas cartesianas en polares de la siguiente manera:

$$g = \sqrt{g_x^2 + g_y^2}$$

$$\theta = arctan\frac{g_y}{g_x}$$

Una vez calculado el gradiente se divide la imágen en celdas del mismo tamaño. Las orientaciones del gradiente se agrupa en intervalos, y se representa cada celda mediante un histograma.Es tipo fijar el tamaño de bin a 9, lo cuál serán 20 grados por bin (360/9 = 20). Posteriormente las celdas se agrupan en bloques, donde cada bloque se obtiene mediante una ventana deslizante cuyo desplazamiento y tamaño deben ser fijados al inicializar el descriptor. Estos histrogramas obtenidos se utilizan para obtener un histograma normalizado de cada celdas. La suma de todos los histogramas es el descriptor denominado HOG.

2.0.2. Parámetros del descriptor

Las imágenes de kmnist son de un tamaño similar a mnist (28x28). Los siguientes parámetros se adaptan correctamente a un tamaño de imágen 28x28 y son los recomendado en la práctica para el caso de MNIST.:

- Tamaño de ventana 28x28: este es el tamaño de las imágenes que se procesarán.
- Tamaño de celda 4x4: por tanto, el área de cada celda, formando un grid de 7x7 celdas en la imagen original. Se calculará un histograma para cada celda.
- Tamaño de bloque 8x8: cada bloque estará formado por 4 celdas completas, es decir, abarca un área de 8x8 píxeles.
- Desplazamiento 2x2: los bloques se moverán con un paso de 2 píxeles en ambas direcciones generando solapamiento entre bloques adyacentes.
- Número de bins 9: valores que formarán el histograma con las magnitudes acumuladas de gradientes en direcciones específicas.

HOG obtiene un descriptor en forma de vector, que representa un histograma similar al representado en las figuras 2.1 y 2.2.

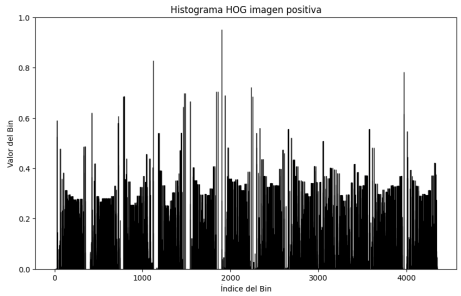

Figura 2.1: Descriptor Hog figura 1.2

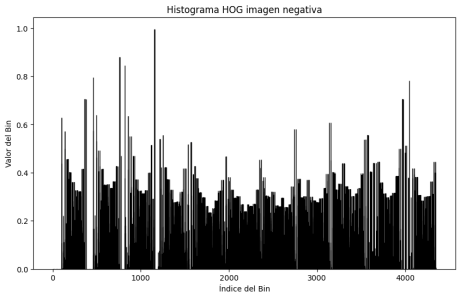

Figura 2.2: Descriptor Hog figura 1.3

Será utilizado el descriptor HOG junto con una red SVM para realizar un ejercicio de clasificación entre las dos clases escogidas. Pasos a seguir:

- Lectura del dataset de entrenamiento con clases negativas y positivas.
- Inicialización de la clase descriptor con los parámetros fijados.
- Búsqueda de hiper-parámetros del modelo SVM con los datos de entrenamiento representados con la clase descriptor.

- Evaluación de la bondad en la clasificación.

El modelo SVM escogido será utilizado junto a los descriptores posteriores para la clasificación.

2.0.3. Búsqueda de hiperparámetros

Se realiza una búsqueda del mejor modelo que combine SVM + HOG, probando diferentes hiper-parámetros de SVM (los descriptores HOG siempre usarán los parámetros de la sección 2.0.2). Las alternativas se comparan utilizando la implementación de GridSearch de Scklearn, que dado un grid de parámetros realiza un *CrossValidation* del tamaño indicado para cada combinación posible y calculá el valor medio obtenido por la misma de la métrica deseada. Realizados todos los ajustes se obtiene la mejor combinación de parámetros de grid. Debido a los elevados tiempos de computación referentes al entrenamiento de redes SVM, especialmente con kernels **rbf** (aproximadamente 5 minutos por fold), se organiza el experimento de búsqueda de hiper-parámetros de la siguiente manera:

- Paso 1: comparación de kernels polinómico (grado 3) y lineal con C=[0.1,1,5,15]. Se realiza un 5-fold cross validation. **Resultado**: modelo con kernel polinómico y C = 0.1.
- Paso 2: Comparación con kernel rbf de parámetros C = [0.1,1,5,15] y gamma = [1, 0.1, 0.01 , 0.001]. En este caso se utilizan solo la mitad de los ejemplos y un cross validation de 3 folds, debido a los altos costes de compute de la búsqueda de hiper-parámetros en este caso. **Resultado**: kernel rbf con C=5 y gamma = 0.01.
- Paso 3: Comparación de los vencedores de ambos casos. Repetimos el experimento comparando finalmente kernels polinómicos y rbf con C = [1,5] y gamma fijado a 0.01. **Resultado**: el mejor modelo encontrado es el modelo de kernel polinómico y C=1.

2.0.4. Entrenamiento del modelo

Fijados los parámetros del modelo SVM a utilizar, son leídas las 6000 imágenes positivas y las 6000 negativas de entrenamiento. Para cada imágen, son calculados los descriptores HOG y almacenados junto con su etiqueta. Estos son los valores usados para el entrenamiento del modelo SVM. Este entrenamiento será realizado de la misma manera con el resto de descriptores de la práctica.

	Dummy	SVM+HoG
Accuracy	0.5	0.998
Precision	0.0	1.0
Recall	0.0	0.997
F1	0.0	0.998
Confusion_matrix	[[100,0] [100,0]]	[[100,0] [3,997]]

Tabla 2.1: Métricas evaluación SVM+HOG

2.0.5. Evaluación del modelo

Similar al conjunto de entrenamiento, tenemos 1000 imágenes positivas y 1000 imágenes negativas a las se calcula un descriptor. El modelo entrenado de la sección 2.0.4 es utilizado para realizar las predicciones, y estas son comparadas con los valores reales de cada imágen. Para determinar que los resultados del modelo provienen de aprendizaje y no del azar, utilizamos para la comparativa un *DummyModel* que siempre predice la clase más abundante.

La tabla 2.1 muestra los resultados de las predicciónes realizas teniendo en cuenta las siguientes métricas:

- **Accuracy**: mide la capacidad de acierto del modelo. Con un valor mayor a 0.99, el modelo comete predicciones errones en menos del 0.01 % de las ocasiones. El modelo dummy solo cuenta con un 0.5 % ya que siempre predice el mismo valor y las clases estaban balanceadas.
- **Precision**: la cantidad de valores positivos que el modelo es capaz de capturar. En este caso el modelo acierto todos los ejemplos positivos mientras que el modelo dummy ninguno.
- **Recall**: mide como de frecuentes son correctas las predicciones positivas. SVM+HOG con un 0.99 % encuentra prácticamente todas las instancias positivas, pero existen 3 falsos negativos.
- **F1**: Mide el balance entre precisión y recall, siendo este casi perfecto para el modelo.
- **Confusion matrix**: el modelo SVM+HOG solo se equivoco clasificando 3 imágenes como negativas que eran positivas.

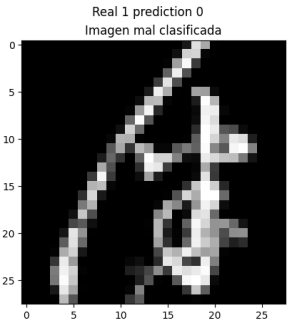

Figura 2.3: Error FN

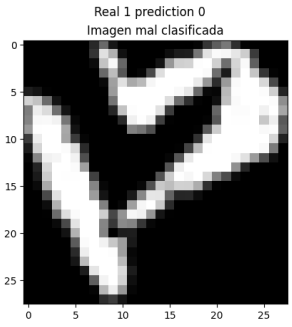

Figura 2.4: Error FN

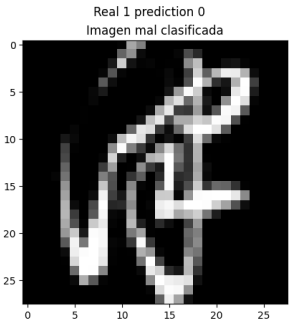

Figura 2.5: Error FN

Los 3 errores en la clasificación son mostrados en las figuras 2.3, 2.4 y 2.5. Respecto a la forma común del carácter a predecir *tsu* (ver figura 1.2), las imágenes 2.3 y 2.5 corresponden a representaciones muy diferentes del carácter que es posible que se vean menos representadas en el dataset (todas las representaciones pueden verse en 1.1). El caso de 2.4, si contiene una representación más similar a la común y por ende un error que podría considerarse más grave.

En conclusión podemos determinar que el modelo SVM entrenado con descriptores HOG tiene una alta capacidad para clasificar entre estos carácteres.

Capítulo 3

LBP: Local Binary Pattern

3.0.1. Introducción

En el siguiente capítulo se realiza la implementación de descriptores LBP para la posterior clasificación de carácteres Hiragana. Los descriptores LBP miden la intensidad de un píxel respecto al vecindario del mismo. Se utilizan ventanas de tamaño 3x3 píxeles (8 píxeles de vecindario más el píxel central). El valor de LBP es calculado asignando a cada miembro del vecindario un valor 0, si su intensidad es menor al pixel central u 1 si su intensidad es mayor. Con los valores 0-1 obtenidos se forma un número binario cuyo valor decimal es asignado al píxel. Con los valores de LBP obtenidos para cada píxel se forma el histograma que es usado posteriormente como descriptor. La figura 3.1 es una representación de valores aleatorios generados con numpy. En 3.2 encontramos cuál sería la representación LBP de dicha figura con la implementación usada. Observamos que esta representación incluye un marco debido a que los valores de las filas/columnas esquina no tiene vecindario.

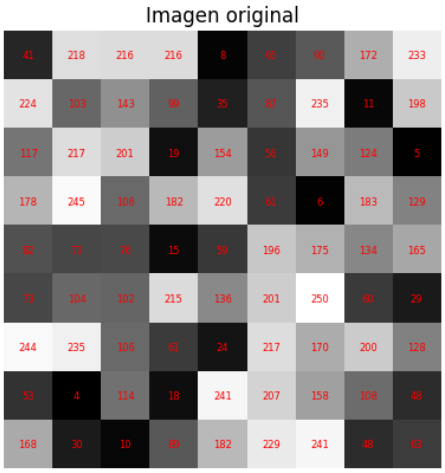

Figura 3.1: Imágen artificial con valores aleatorios.

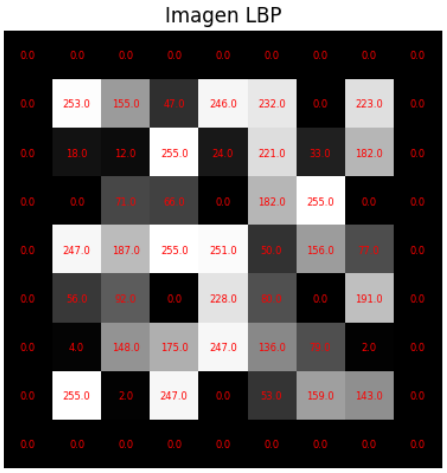

Figura 3.2: Representación valores LBP 3.1.

En la figura 3.4 se visualizan los valores que toma el vecindario del primer píxel tomado en cuenta por LBP (figura 3.3. Estos se ordenan siguiendo las agujas del reloj desde la esquina superior derecha. En este caso 11111101, como su representación en base decimal 253 este es el valor tomado por el píxel.

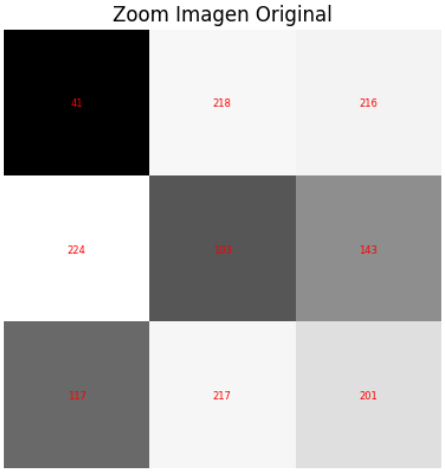

Figura 3.3: Zoom vecindario LBP usado para el primer píxel 3.1.

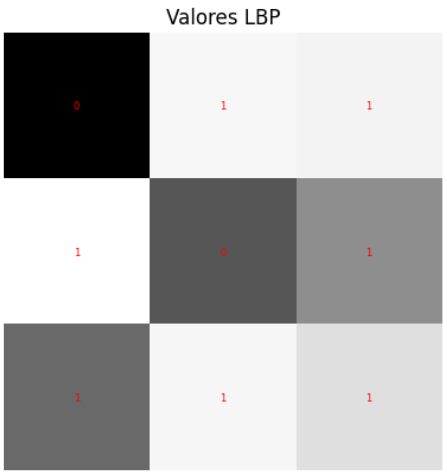

Figura 3.4: Representación *threshold* LBP 3.3

En las figuras 3.5 y 3.7, se observa la representación LBP para dos ejemplos de las clases positiva y negativa respectivamente. Junto a ellas sus respectivos histogramas de valores 3.6 y 3.8.

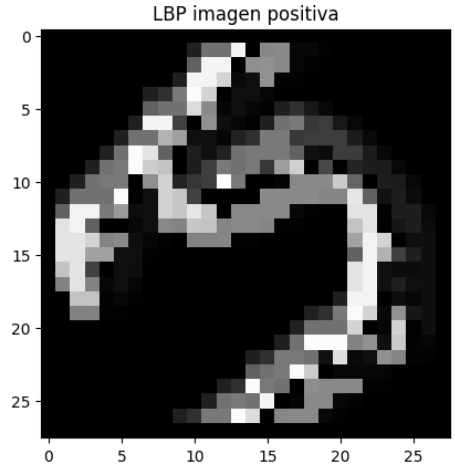

Figura 3.5: Imágen LBP para figura 1.2

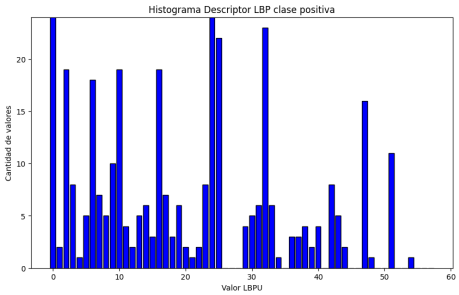

Figura 3.6: Histograma de valores3.5

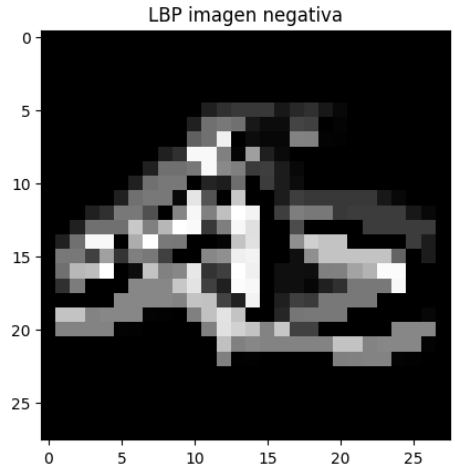

Figura 3.7: Imágen LBP para figura 1.3

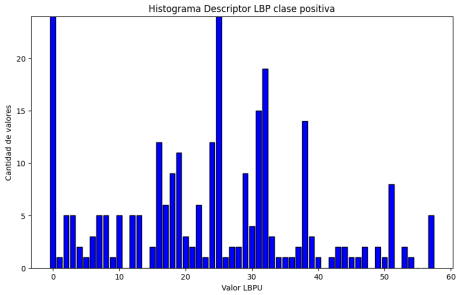

Figura 3.8: Histograma de valores 3.7

3.0.2. Implementación.

En esta sección se abordan los detalles básicos sobre la implementación de LBP utilizada. El descriptor se encuentra en la clase LBPDesciptor y consta de los siguientes métodos.

```
def __init__(self, window_size: int = 3):
    """
    Inicializacion de los parametros de entrada.
    Realmente el codigo esta solo preparado para
    implementacion basica.
    """
    self._window_size = window_size
    self._border = window_size // 2
```

Figura 3.9: Función de inicialización LBPDescriptor.

El código 3.9 muestra la inicialización de parámetros. Realmente la implementación está preparada para diferentes valores de ventana menos en su método de computación del valor LBP de un píxel donde se *hardcodea* el orden utilizado para calcular el valor binario.

```
    def _binary_neighborhood_comparation(self,
    window_list_values: list, pixel_value: int):
        """
        Para una ventana en formato lista de valores y el valor
        del pixel central, se calcula el valor decimal
        correspondiente al valor binario resultante de LBP.

        ...

        Attributes
        ---
            window_list_values : list
                Valores de la ventana a calcular.

            pixel_value: int
                Valor del pixel central.

        ...

        Returns
        ---
            Devuelve el valor LBP asociado al pixel central.
        """
        values = [1 if pixel > pixel_value else 0 for pixel in
    window_list_values]
        values_without_center = [values[i] for i in [2, 5, 8,
    7, 6, 3, 0, 1]]
        binary_value = int("".join(str(a) for a in
    values_without_center), 2)
        return binary_value
```

Figura 3.10: Calculo del valor LBP de un píxel.

Es computado el valor de LBP de un píxel a través de los valores de su vecindario 3.10. Para ello se compara el valor del píxel central con el resto de valores y se fija a 1 aquellos vecinos mayores que el píxel central y 0 para los menores o iguales.

```
1
2       def _calculate_windows_list_format(self, img: np.ndarray,
        pixel: Pixel):
3           """
4           Calcula para una imagen dada y un pixel determinado su
5           ventana de vecinos.
6
7           ...
8
9           Attributes
10          ---
11              img : np.ndarray
12                  imagen original.
13              pixel : Pixel
14                  pixel central al que calcular el valor.
15
16          Returns
17          ---
18              Devuelve la ventana window_size*window_size
        correspondiente al pixel central.
19
20          """
21          combinaciones_x_y = [(x, y)
22                               for y in range(pixel.y - self.
        _border, pixel.y + self._border + 1)
23                               for x in range(pixel.x - self.
        _border, pixel.x + self._border + 1)
24                               ]
25          return [img[y, x] for x, y in combinaciones_x_y]
```

Figura 3.11: Cálculo de un vecindario en formato lista.

Se utiliza 3.0.2 para calcular la lista de vecinos de un determinado píxel, creando una lista 9x9 que devuelve la ventana que incluye ambos (vecindario y píxel):

$$\begin{bmatrix} x_1y_1 & x_1y_2 & x_1y_3 \\ x_2y_1 & x_2y_2 & x_2y_3 \\ x_3y_1 & x_3y_2 & x_3y_3 \end{bmatrix}$$

La transformación de esta matriz en un vector fila sería:

$$[x_1y_1, x_1y_2, x_1y_3, x_2y_1, x_2y_2, x_2y_3, x_3y_1, x_3y_2, x_3y_3]$$

```
    def compute_lbp_image(self, img: np.ndarray):
        """
        Calcula la imagen lbp correspondiente a la imagen original.

        Attributes
        ---
            img : np.ndarray
                imagen original.

        Returns
        ---
            Imagen LBP asociada.

        """
        rows, columns = img.shape
        indexs = [Pixel(x, y) for y in range(self._border, rows - self._border) for x in
                  range(self._border, columns - self._border)]

        pixel_value_with_windows = [(img[p.y, p.x], self._calculate_windows_list_format(img, p)) for p in indexs]

        lbp_image = np.float32(
            [self._binary_neighborhood_comparation(window_list_values=w[1], pixel_value=w[0]) for w in
             pixel_value_with_windows])

        return lbp_image
```

Figura 3.12: Computación de una imagen en LBP.

La función representada en 3.12 toma una imágen, calcula para los pixeles no correspondientes al borde su vecindario usando la función 3.0.2. Finalmente para cada vecindario se computa el valor de LBP con la función 3.10.

```
    def compute(self, img: np.ndarray):
        """
        Calcula el histograma LBP y por tanto el descriptor de
    una imagen.

        Attributes
        ---
            img : np.ndarray
                imagen original.

        Returns
        ---
            Descriptor LBP de la imagen asociada.
        """
        lbp_img = self.compute_lbp_image(img)
        lbp_descriptor = np.float32(
            [len(np.where(lbp_img == value)[0]) for value in
    range(0, 256)]
        )
        return lbp_descriptor
```

Figura 3.13: Cálculo del histograma.

El paso final consiste en tomar la imágen, calcular su representación lbp (vease 3.12) y para cada valor posible de intensidad (de 0 a 255) calcular el número de píxeles con este valor. Los 256 forman un histograma con la frecuencias en las que aparece cada valor de intensidad.

3.0.3. Evaluación.

	Dummy	**SVM+HOG**	**SVM+LBP**
Accuracy	0.5	0.9985	0.86
Precision	0.0	1.0	0.83
Recall	0.0	0.997	0.919
F1	0.0	0.998	0.87
Confusion_matrix	[[100,0] [100,0]]	[[100,0] [3,997]]	[[815,185] 81,919]]

Tabla 3.1: Evaluación de modelos.

Respecto a los modelos anteriormente testeados, SVM+LBP obtiene los siguientes resultado:

- **Accuracy**: el modelo continua teniendo un porcentaje alto de acierto de 0.86 % aunque disminuye respecto SVM con descriptores HOG.

- **Precision**: En este caso el modelo pierde cierta capacidad para predecir las imágenes positivas con precisión, clasificando como estas 185 imágenes que realmente son negativas.
- Recall: También disminuye el total de imágenes positivas encontradas, pasando de 3 errores a 81.
- **F1**: Debido a los cambios en precision y recall el valor de f1 también se ve afectado.
- **Confusion matrix**: SVM+LBP aumenta de 3 a 107 en falsos negativos y de 0 a 185 en falsos positivos.

Las figuras 3.14 y 3.15 son algunos de los errores cometidos por SVM+LBP. En este caso al ser mucho mayor el número de errores sacar conclusiones puede ser precipitado.

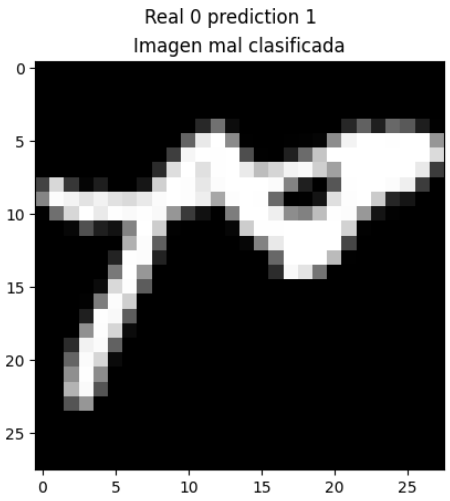

Figura 3.14: Imágen mal clasificada por SVM+LBP

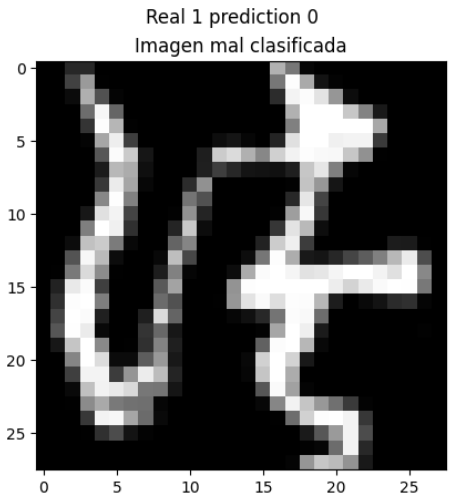

Figura 3.15: Imagen mal clasificada por SVM+LBP

Capítulo 4

LBP uniforme

4.0.1. Introducción

Un patrón binario local se denomina uniforme si al recorrer el su patrón este contiene como máximo dos transiciones. Se llaman transacciones al paso bit a bit de valor 0 a 1 o viceversa.

00010011 $\rightarrow$ 4 transacciones:

- 00***01***0011
- 000***10***011
- 00010***01***1
- ***0***0010011

Cuando un patrón contiene 2 o menos transacciones se denomina uniforme. Para el cálculo de etiquetas LBP uniformes, es asignado a cada píxel un valor similar a LBP y calculado si su patrón es uniforme. Caso de que sí, asignamos el valor decimal correspondiente al patrón. En caso de que no, se asigna una etiqueta -1 (podría ser cualquier otro valor). Finalmente el histograma descriptor de la imágen solo contendrá aquellos valores con patrones uniformes (ver tabla 4.1).

Tabla 4.1: Tabla de patrones uniformes

Binario	**Decimal**	**Binario**	**Decimal**
00000000	0	10000000	128
00000001	1	10000001	129
00000010	2	10000011	131
00000011	3	10000111	135
00000100	4	10001111	143
00000110	6	10011111	159
00000111	7	10111111	191
00001000	8	11000000	192
00001100	12	11000001	193
00001110	14	11000011	195
00001111	15	11000111	199
00010000	16	11001111	207
00011000	24	11011111	223
00011100	28	11100000	224
00011110	30	11100001	225
00011111	31	11100011	227
00100000	32	11100111	231
00110000	48	11101111	239
00111000	56	11110000	240
00111100	60	11110001	241
00111110	62	11110011	243
00111111	63	11110111	247
01000000	64	11111000	248
01100000	96	11111001	249
01110000	112	11111011	251
01111000	120	11111100	252
01111100	124	11111101	253
01111110	126	11111110	254
01111111	127	11111111	255

Para la imagen artificial vista en la sección 3.0.1, la representación LBP uniforme correspondiente puede visualizarse en la figura 4.2. Se observa que en esta ocasión son muchos menos los valores representados en la imágen ya que todas las etiquetas no uniformes se corresponden con -1.

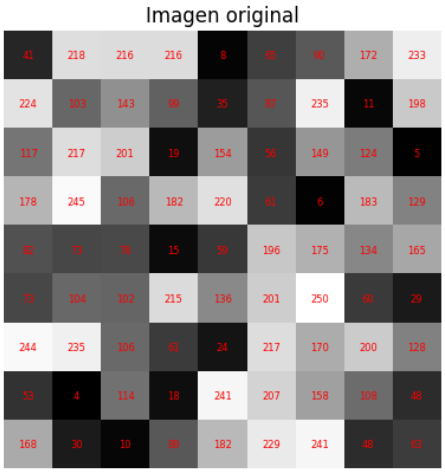

Figura 4.1: Imagen artificial con valores aleatorios.

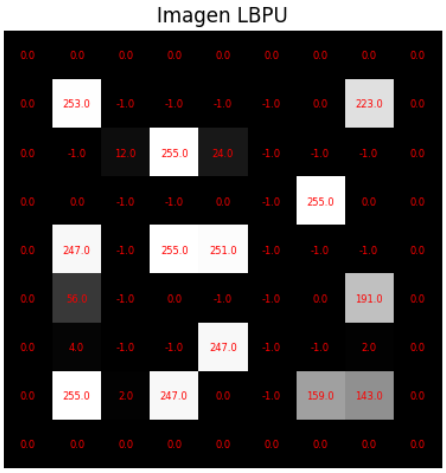

Figura 4.2: Representación valores LBPU 3.1.

Para el vecindario del pixel x=1 e y=5 (vease la figura 4.3) la representación binaria de LBP uniforme 00111000 contiene dos transicciones y por ende es asignada la etiqueta representada bajo su valor binario 56.

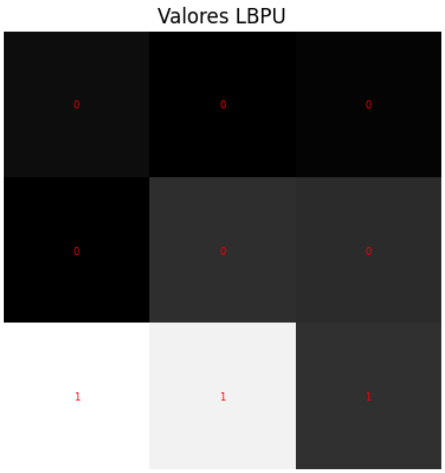

Figura 4.3: Zoom vecindario LBP Uniforme usado para el píxel (1,5) 4.2.

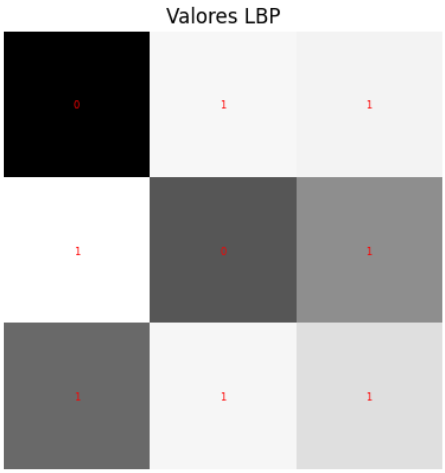

Figura 4.4: Representación threshold LBP Uniforme 4.3

Los descriptores correspondientes a las imágenes positiva y negativa pueden visualizarse en las figuras 4.5 y 3.7 junto con sus descriptores 4.8 y 4.8, mucho más compactos que los vistos para LBP.

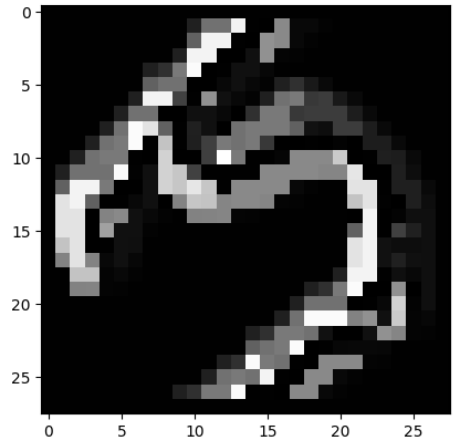

Figura 4.5: Imágen LBPU para figura 1.2

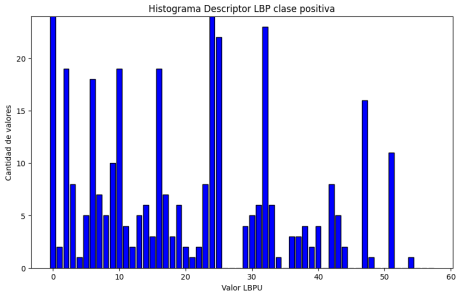

Figura 4.6: Histograma de valores 4.5

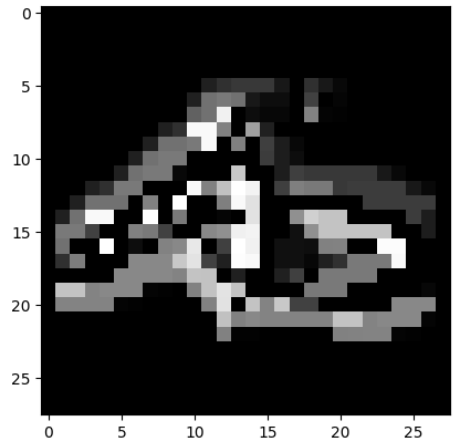

Figura 4.7: Imágen LBPU para figura 1.3

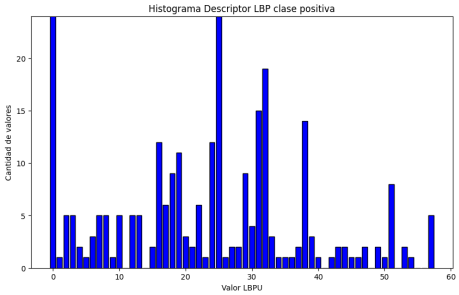

Figura 4.8: Histograma de valores4.7

4.0.2. Implementación.

Similar a la vista en la sección 3.0.2 a excepción de los métodos de computación de la imágen y el histograma.

```
1     def _uniform_neighborhood_comparation(self,
      window_list_values: list, pixel_value: int) -> int:
2         """
3         Para una ventana en formato lista de valores y el valor
       del pixel central, se calcula el
4         valor decimal
5         correspondiente al valor binario resultante de LBP
      Uniforme.
6
7         ...
8
9         Attributes
10        ---
11            window_list_values : list
12                Valores de la ventana a calcular.
13
14            pixel_value: int
15                Valor del pixel central.
16
17        ...
18
19        Returns
20        ---
21            Devuelve el valor LBP Uniforme asociado al pixel
      central
22        """
23        values = [1 if pixel > pixel_value else 0 for pixel in
      window_list_values]
24        values_without_center = [values[i] for i in [2, 5, 8,
      7, 6, 3, 0, 1]]
25        values_changing = [value for index, value in enumerate(
      values_without_center) if value != values_without_center[
      index - 1]]
26        if len(values_changing) <= 2:
27            label = int("".join(str(a) for a in
      values_without_center), 2)
28        else:
29            label = -1
30
31        return label
```

Figura 4.9: Computación de un valor LBP Uniforme.

En la computación del valor de un pixel 4.9 contabiliza el número de transiciones y para aquellas cuyo valor es mayor a dos se asigna la etiqueta estática -1.

```
        def compute(self, img):
        """
        Calcula el histograma LBP Uniforme y por tanto el descriptor de una imagen.

        Attributes
        ---
            img : np.ndarray
                imagen original.

        Returns
        ---
            Descriptor LBP Uniforme de la imagen asociada.
        """
        lbpu_img = self.compute_lbpu_image(img)
        lbpu_descriptor = np.float32(
            [len(np.where(lbpu_img == value)[0]) for value in POSSIBLE_UNIFORM_VALUES]
        )
        return lbpu_descriptor


POSSIBLE_UNIFORM_VALUES = [0, 1, 2, 3, 4, 6, 7, 8, 12, 14, 15, 16, 24, 28, 30, 31, 32, 48, 56,
60, 62, 63, 64, 96, 112, 120, 124, 126, 127, 128, 129, 131, 135, 143, 159, 191, 192, 193, 195,
199, 207, 223, 224, 225, 227, 231, 239, 240, 241, 243, 247, 248, 249, 251, 252, 253, 254, 255]
```

Figura 4.10: Computación histograma LBPU.

En el código de computación del histograma 4.10 solo mantenemos los valores uniformes, obteniendo descriptores de 58 valores.

4.0.3. Evaluación

	Dummy	**SVM+HOG**	**SVM+LBP**	**SVM+LBP**
Accuracy	0.5	0.9985	0.86	0.86
Precision	0.0	1.0	0.83	0.83
Recall	0.0	0.997	0.919	0.91
F1	0.0	0.998	0.87	0.87
Confusion_matrix	[[100,0] [100,0]]	[[100,0] [3,997]]	[[815,185] 81,919]]	[[813,187] 87,913]]

Tabla 4.2: Resultados Test de todos los modelos.

LBP uniforme consigue resultados iguales a LBP con descriptores más pequeños. Solo 8 más son las imágenes que LBP consigue clasificar de manera

correcta respecto LBP uniforme.

Capítulo 5

Búsqueda de objetos

En este apartado abarca la tarea de detección de un carácter tsu dentro de un conjunto de caracteres (imágen 5.1). Se utilizan los descriptores LBP uniformes, con el modelo SVM entrenado con la imágenes *train* de las clases 3 y 7. Estos son computados por una ventana deslizante por todas las imágenes de una pirámide de escalas. De cada ventana obtenemos devolvemos como ventana de reconocimiento del objeto aquella donde la predicción hecha por SVM+LBP uniforme contenía una mayor probabilidad de pertenecer a la clase 3 (tsu).

Figura 5.1: Imágen con múltiples carácteres.

5.0.1. Implementación.

La función **??** muestra como se obtiene la imágen más probable a partir de una ventana deslizante.

```
def extract_better_result(og_img, model, descriptor_model,
    step_size=28, window_size=(28, 28)):
    best_margin_descriptor_result = 0.0
    best_window = None
    og_img_frame = None
    for window_margin_tuple in extract_windows(og_img,
    window_size=window_size, step_size=step_size):
        window_img = window_margin_tuple[0]
        new_descriptor = descriptor_model.compute(window_img)
        distance_descriptor_margin = model.predict_proba(
    new_descriptor.reshape(1,-1))[0][0]
        if distance_descriptor_margin >
    best_margin_descriptor_result or best_window is None:
            best_margin_descriptor_result =
    distance_descriptor_margin
            best_window = window_img
            og_img_frame = window_margin_tuple[1]

    return best_window, best_margin_descriptor_result,
    og_img_frame
```

Figura 5.2: Extracción de la ventana más probable.

El código 5.3 calcúla la ventana más probable en diferentes escalas, las cuales se contruyen mediante un factor de escalado que se aplica hasta llegar a un tamaño mínimo.

```
def pyramid(image, scale=1.5, minSize=(28, 28)):
    yield image
    while True:
        w = int(image.shape[1] / scale)
        image = imutils.resize(image, width=w)
        if image.shape[0] < minSize[1] or image.shape[1] <
    minSize[0]:
            break
        yield image
```

Figura 5.3: Pirámide de imágenes.

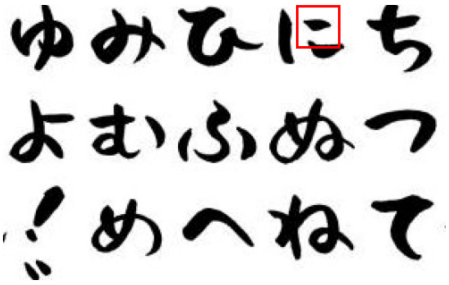

Figura 5.4: Escala normal

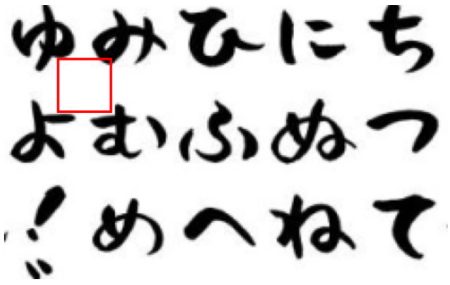

Figura 5.5: Escalado 1

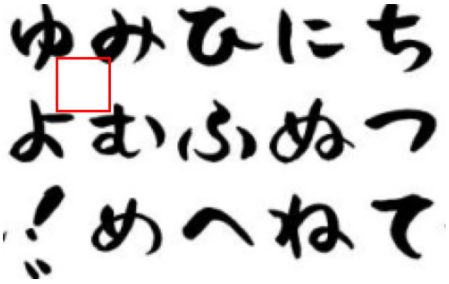

Figura 5.6: Rescalado 2

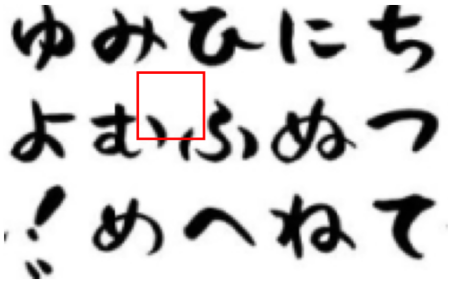

Figura 5.7: Escalado 3

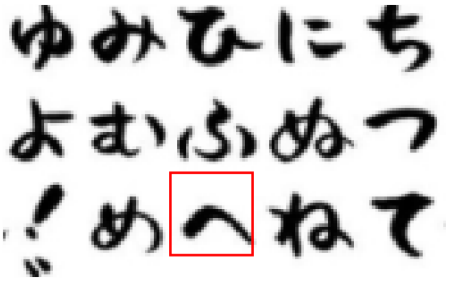

Figura 5.8: Escalado 4

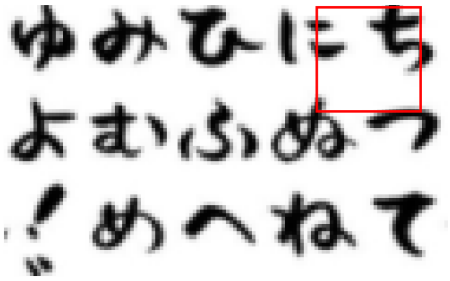

Figura 5.9: Escalado 5

En el escalado 4 5.8 donde el tamaño de la ventana tiene el tamaño del carácter es encontrado el carácter tsu.

Bibliografía

[1] T. Clanuwat, M. Bober-Irizar, A. Kitamoto, A. Lamb, K. Yamamoto, and D. Ha. (2018) Deep learning for classical japanese literature.

[2] (2024). [Online]. Available: https://es.wikipedia.org/wiki/Hiragana